Assignment 2

Team number: 2
Team members

Name	**Student Nr.**	**Email**
Lucas Ponticelli	2687740	l.p.i.ponticelli@student.vu.nl
Khagan Mammadov	2669723	k.mammadov@student.vu.nl
Jeroen Klaver	2696798	j.a2.klaver@student.vu.nl
David Breitling	2685269	d.k.l.breitling@student.vu.nl

Format: Class and package names are **bold**; attributes, operations, and associations are underlined text; objects are *italic*.

Feedback from Assignment 1

Due to the positive feedback we received on our first assignment, we only had to revise a few minor issues pointed out by our TA. As such, our revised version of Assignment 1 is nearly identical to our initial submission. One of the changes we made was making it more clear in our writing that biometrics are optional to the user and not as a whole. We now also correctly indicate that file size limits are reliability, not performance or efficiency attributes. The final change we made was correctly indicating the total amount of hours spent on the assignment in our time plan.

Introduction

Author: David Breitling

In order to attain a better understanding of how to achieve our goal stated in the previous iteration (Assignment 1), i.e. “to visualise and process sport activities exported in the GPX format from sport tracking applications, such as Strava and RunKeeper”, we first brainstormed as a team. During this brainstorming session, we came up with a general overview of the system before making any formal UML diagrams. Considering the fact that the feedback on our previous iteration was positive in every respect, we were confident that we would go in the right direction if we based our general overview on the simple flowchart provided in Assignment 1. The creation of this general overview helped us with visualising the rough roadmap of our project and laying down the framework for our application. The formal diagrams further below in this document describe the inner workings of our system to a greater extent and in more detail.

The informal general system overview diagram can be found here:
https://imgur.com/DvYocMB

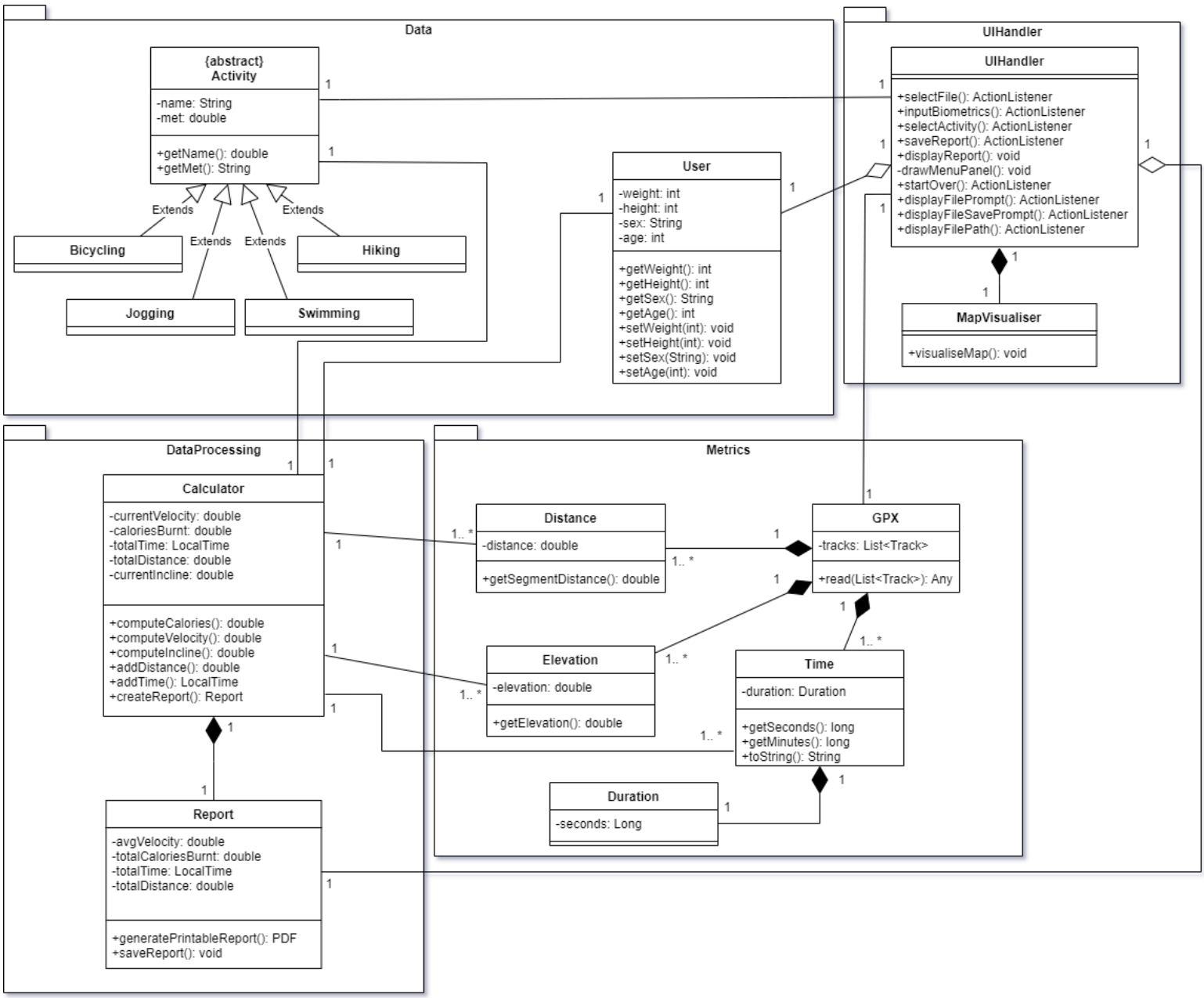
Implemented features

ID	**Short name**	**Description**
F1a	Import	The tool shall be able to import a GPX file.
F2	Sport Activity	The tool shall be able to assign a sport activity to a GPX file.
F4	Compute metrics	The tool shall be able to compute metrics based on the imported GPX file, such as: ● Distance ● Speed ● Time ● Elevation ● Calories burned

Used modelling tool: draw.io

Class diagram

Authors: Lucas Ponticelli, Jeroen Klaver



Package: Data

<abstract> Activity

Activity is an abstract class with 2 attributes, and 2 operations. It is extended currently by 4 other classes which are **Bicycling**, **Jogging**, **Swimming**, **Hiking** as seen on the diagram. The **Activity** class is related to the **UIHandler** and the **Calculator** by means of a bidirectional relationship. It represents the different activities that are currently supported by our app, and is modular for future development.

Attributes

- name: String – The name of the activity that is assigned in the extending classes of **Activity**
- met: double – The metabolic equivalent of a task used to accurately calculate the amount of calories used in different sports

Operations

- getName(): String – Used to access the name of the activity outside of the **Activity** class
- getMet(): double – Used to access the met of the activity in question outside of the **Activity** class

User

User is a class with 4 attributes and 8 getter and setter operations. This class has a shared aggregation with the **UIHandler** class but does not have access to its attributes, whereas **UIHandler** has access to **User**. **User** also has a bi-directional relationship with the **Calculator** class as it contains the different input data from the user.

Package: UIHandler

UIHandler

UIHandler is a class in charge of handling all the UI in our application, such as the interface to select the gpx file the user would like to import, as well as the interface for the user to input their data for a more precise calculation of their calories burnt during their workout. It has a composition relationship with the **MapVisualiser** class in the same package as the **MapVisualiser** cannot exist without a UI to represent it. It also has bi-directional relationships with the **GPX** and the **Report** classes. Both of these being one to one as we discussed that there could only be one GPX file being parsed in our app at the same time, so the relationship with the UI would be 1 to 1.

Operations

- selectFile(): ActionListener – Operation called when a button is clicked in the UI to select the file indicated by the chosen file path
- inputBiometrics(): ActionListener – Operation called when a button is clicked in the UI to select the biometrics inputted by the user
- selectActivity(): ActionListener – Operation called when a button is clicked in the UI to select the activity chosen by the user
- saveReport(): ActionListener – Operation called when a button is clicked in the UI to save a report with a summary of the user's computed metrics and a map
- displayReport(): void – Operation called by the application once the metrics have been calculated and can be displayed
- drawMenuPanel(): void – Operation called at the launch of the app to display the menu of our application
- startOver(): ActionListener – Operation called when a button is clicked in the UI to start over the different steps to generate a new report with new parameters
- displayFilePrompt(): ActionListener – Operation called when a button is clicked in the UI to prompt the user to choose a gpx file
- displayFileSavePrompt(): ActionListener – Operation called when a button is clicked in the UI to prompt the user to choose where to save a PDF file
- displayFilePath(): ActionListener – Operation called to display the path of the currently selected file

MapVisualiser

MapVisualiser is a class in charge of displaying the map with the track traced on the map relevant to its position. It only has a relationship with the **UIHandler** explained above.

Operations

- visualiseMap(): void – Operation called by the application once the report has been generated by the **Calculator** class with createReport()

Package: DataProcessing

Calculator

Calculator is a class in charge of all the calculations in our application, such as calculating the calories burnt between two waypoints to then add them up and eventually get a total calories burnt value for the workout. It has 3 bi-directional relationships with classes in the **Metrics** package namely **Distance**, **Elevation** and **Time**. All of these relationships are required to calculate the different values to then use for the calculation of calories. As there can be one or many waypoints, we decided to make the relationships one calculator to one-to-many distances, elevations or times. Another relationship is with the **Report** class which is a composition as a

report may not exist in our application without a calculator creating one, only a single report may exist per calculator instance too thus the 1 to 1 relationship.

Attributes

- currentVelocity: double – Attribute to save the current velocity of the user between waypoints to be then added and divided to get the average velocity throughout the workout
- caloriesBurnt: double – Attribute to save the calories burnt between waypoints and then added up to get the total calories burnt in their workout
- totalTime: LocalTime – Attribute to save the time in between waypoints to calculate the velocity
- totalDistance: double – Attribute to save the distance in between waypoints to calculate the velocity
- currentIncline: double – Attribute to save the calculated incline between waypoints

Operations

- computeCalories(): double – Operation to compute the calories burnt between waypoints
- computeVelocity(): double – Operation to compute the current velocity between waypoints
- computeIncline(): double – Operation to compute the incline between waypoints
- addDistance(): double – Operation to add the total distance of the workout
- addTime(): LocalTime – Operation to add the total time of the workout
- createReport(): Report – Operation to create a new **Report** object and assign the different values calculated from the previous operations

Report

Report is a class that has all the different attributes that will be displayed in the summary report for the user to then save or just view. All of its relationships have been enumerated already.

Attributes

- avgVelocity: double – Attribute to save the average velocity of the user throughout their whole workout
- totalCaloriesBurnt: double – Attribute to save the total amount of calories burnt by the user
- totalTime: LocalTime – Attribute to save the total time the user spent on their workout
- totalDistance: double – Attribute to save the total distance travelled by the user

Operations

- generatePrintableReport(): PDF – Operation called after the report is displayed to the user for them to then save the report if needed
- saveReport(): void – Operation called when the saveReport() function in the **UIHandler** class is called to save the generated PDF report from the previous operation

Package: Metrics

GPX

GPX is a class that is generated as soon as a gpx file is inserted in the application, and it will deal with parsing the file with the help of 3 other classes: **Distance**, **Elevation** and **Time**. They all have a composition relationship with the **GPX** as these classes do not exist without a gpx file. They each have a 1-to-many to 1 **GPX** class relationship too as the gpx file may have one or several waypoints.

Attributes

- Tracks: List<Track> – Attribute to save the different tracks, if there's one or more, to be then parsed into the different relevant information

Operations

- read(List<Track>): Any – Operation to read the different tracks into the relevant classes to be then used for our calculations

Time

Time is a class that will take care of saving all time related information. It has a composition relationship with the **Duration** class as the attribute duration in this class is of type **Duration**.

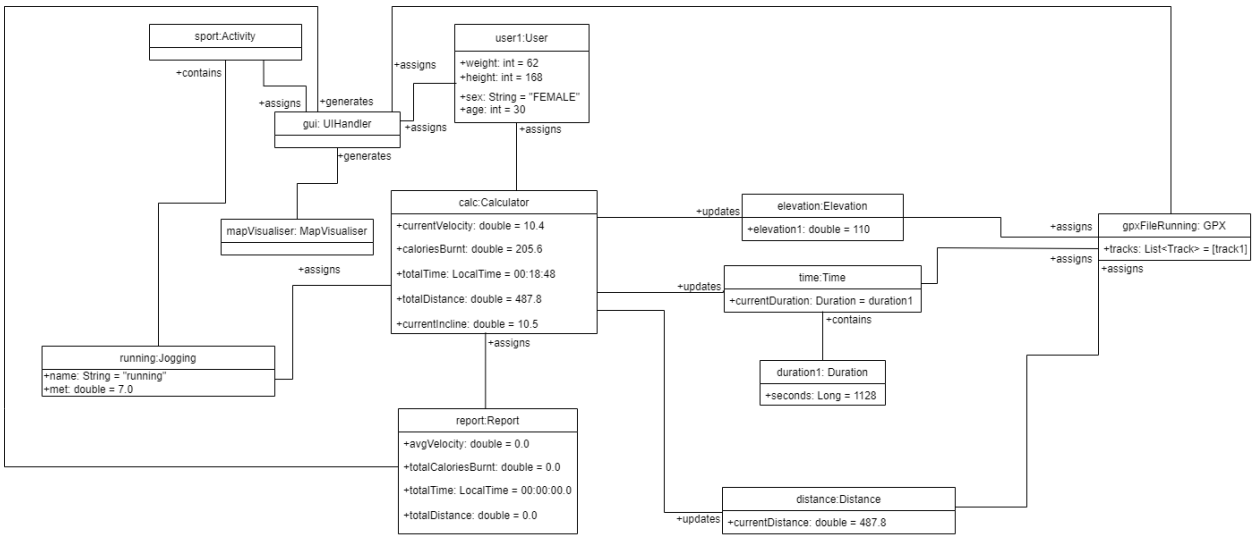
Attributes

- duration: Duration – Attribute to save the time difference between 2 waypoints

An alternative to this would be to implement methods to calculate the values for different types of metrics within **Calculator**, however we decided against this as each metric having its own class keeps the code modular.

Object diagram

Authors: Lucas Ponticelli, Jeroen Klaver



This object diagram displays a “snapshot” of the system during which the system is calculating the metrics that have been given by the chosen GPX file. All of the input variables have already been set by this time and the calculator is looping over the different waypoints inside of the GPX class. An alternative to this would be to use the methods provided by the **GPX** class instead, however we decided against this we could achieve a more accurate estimate of calories burned by calculating the metrics manually.

- **Activity** and **Jogging** class - The **Activity** class is an abstract class containing multiple subclasses of different physical activities the user can select. In this instance the user has selected the “**Jogging**” activity which assigns specific metrics to be used by the **Calculator** class later. These values will be different for each activity.

- **User** class - This class contains the biometric data that can be filled in by the user. If the user does not fill in data then it would use default values. However, in this instance the user has filled in their personal information.

- **GPX**, **Elevation**, **Time** and **Distance** classes - The GPX file contains multiple waypoints that contain the metrics already displayed in the diagram. We then loop over each waypoint and calculate the values in total as well as between each waypoint to be able to calculate values for the final Report class.

- **Calculator** class - This class obtains all its metrics from the **User**, **Jogging**, **Elevation**, **Time** and **Distance** classes to calculate values that can then be displayed to the user in the **Report** class.

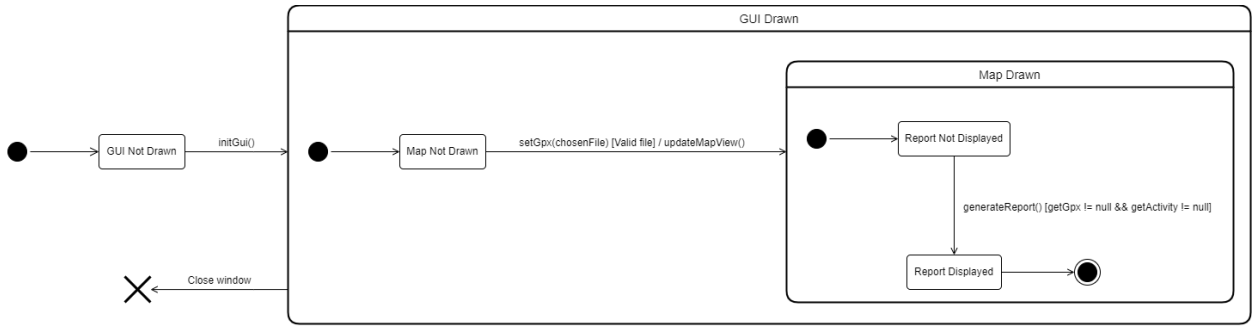
- **Report** class - After **Calculator** is finished it will create an instance of the **Report** class using the calculated values as the variables for the **Report** class. In this instance the **Calculator** instance has not yet finished looping over the GPX file and therefore the report class has yet to be created by the **Calculator**. We chose to display this as a **Report** class with the values set to zero.

- **UIHandler** class - This class contains no variables and only connects to the classes that would require user input. This includes setting user biometrics, setting an activity, setting the GPX file and choosing to save the report.

State machine diagrams

Authors: Khagan Mammadov, David Breitling

State Machine - GUI-User Interaction



See above for a state machine diagram of the **UIHandler** class. The **UIHandler** class is responsible for controlling the interaction of the user with the system and the GUI. Initially, as the application is launched, the *GUI Not Drawn* state is entered. Once the initGui() method is called, the GUI is initialised and the *GUI Drawn* state is entered.

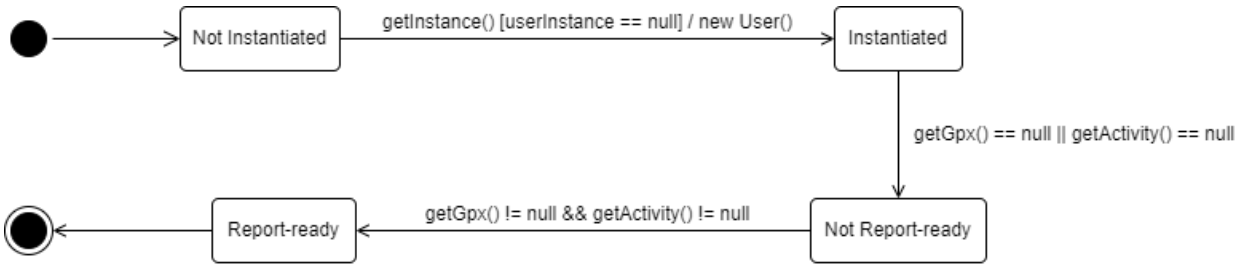
Initially, the *Map Not Drawn* substate is entered as the user has not yet chosen a .gpx file. The transition to the *Map Drawn* state is triggered once a valid .gpx file is chosen. This transition also triggers the updateMapView() method which updates the **MapView** object with data from the chosen .gpx file. Choosing an invalid file causes the GUI to remain in the *Map Not Drawn* state until a valid .gpx file is chosen.

Once the *Map Drawn* substate is entered, the user is presented with a visualisation of the chosen .gpx file. Initially, the machine remains in the *Report Not Displayed* substate. The transition to the *Report Displayed* substate is triggered once the user chooses to generate a report given that they have selected a valid file and activity. The **TextField** object displayed on screen is updated once the generateReport() method is triggered. An alternative to this would

be to automatically generate and display a report without the user explicitly having to press the ‘Generate report’ button. The condition for a valid file and activity would then be checked once a file or activity is chosen. However, we decided to include a button to generate a report, in order to provide a more tactile experience.

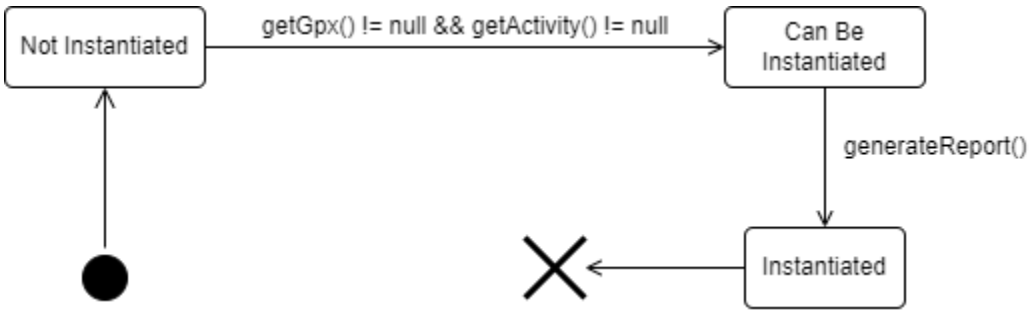
Once the *Report Displayed* substate is entered, the user may choose to remain in this state, close the application, or choose a new .gpx file. If the user chooses a new .gpx file, the state of the machine will not be affected by the validity of the file. This is because an invalid file causes the **MapView** object to not get updated and keep displaying the visualisation of the previous .gpx file. Similarly, not updating the **TextField** object on screen causes the GUI to keep displaying the report of the previous .gpx file. *Map Drawn* and *Report Displayed* ultimately remain the final substates of the machine as the displayed visualisation and report are never cleared. An alternative to this would be to reset the **MapView** and **TextField** objects if an invalid file is chosen, however we decided against it as we felt it was redundant.

State Machine - User



See above for the state machine diagram of the **User** class. From the initial state, the **User** class enters the *Not instantiated* state. In the event that getInstance() is called, the state transitions to the *Instantiated* state. During the transition, if `userInstance` is `null`, `new User()` is executed. In the event that either getGpx() or getActivity() return `null`, the *Not Report-ready* state is entered. In the event that both getGpx() and getActivity() both do not return `null`, the final *Report-ready* state is entered. Note that getGpx() and getActivity() are **User** methods. In the actual implementation, they are being called as User.getInstance().getGpx() and User.getInstance().getActivity() respectively. An alternative to this would be to instantiate **User** objects on demand, however this would falsely imply the possibility of the simultaneous existence of several **User** objects.

State Machine - Report



See above for the state machine diagram of the **Report** class. From the initial state, the **Report** class enters the *Not Instantiated* state. In the event that both getGpx() and getActivity() do not return `null`, the *Can Be Instantiated* state is entered. In the event that generateReport() is called, this state transitions to the *Instantiated* state, from where the instance of **Report** is destroyed. An alternative to this would be to apply the Singleton design pattern to the **Report** class and to set its parameters when needed instead of instantiating a new **Report** object on demand, however we found this to bloat our code with redundant setter operations.

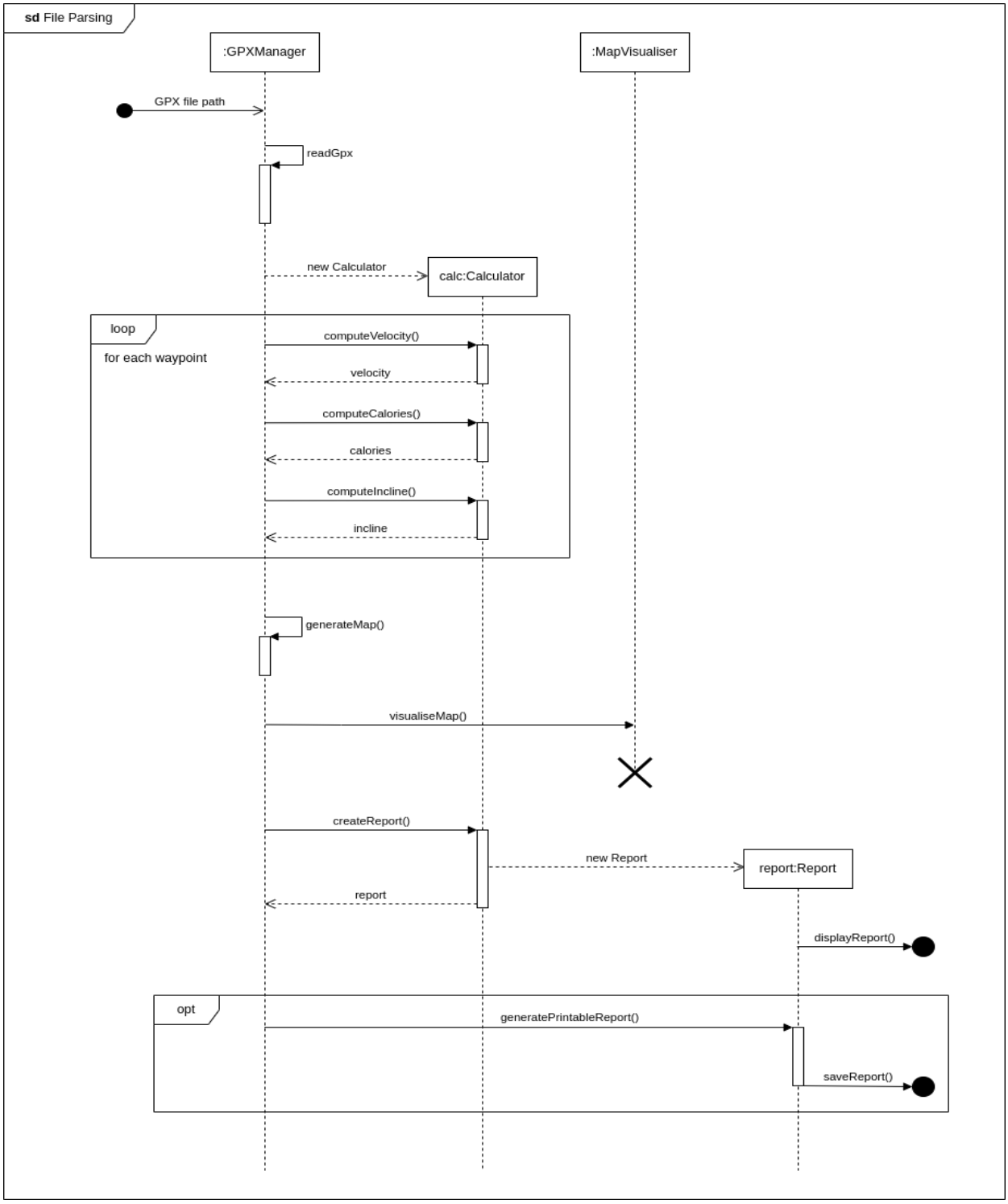
Sequence diagrams

Authors: Khagan Mammadov, David Breitling

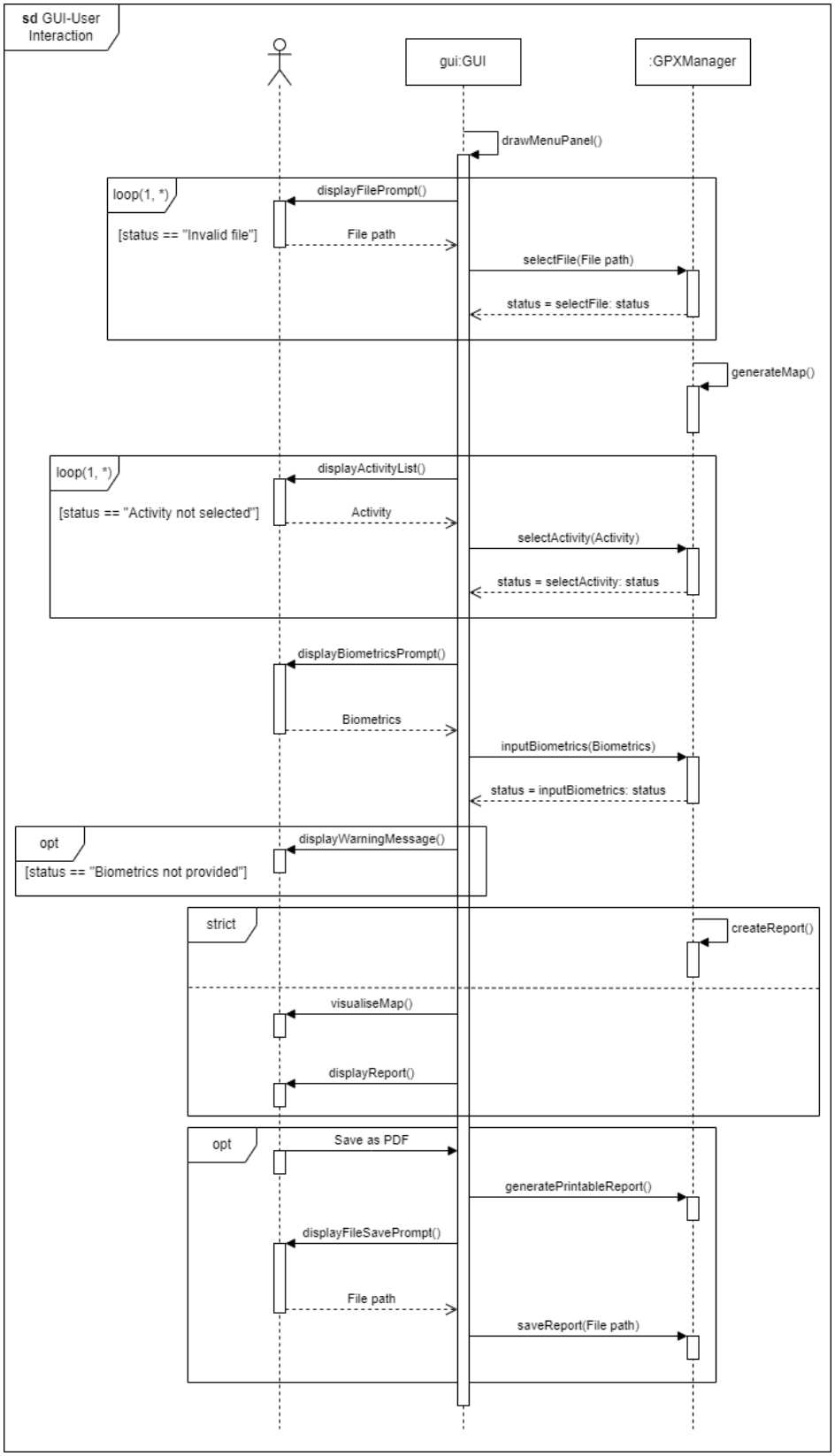
Sequence Diagram - File Parsing

The diagram below details how a file is parsed by our system. Once user input has been collected (see **SD** GUI-User Interaction), the application instance (of **GPXManager**) receives the path to the GPX file. It then does a self-call to readGpx(), which handles the basic parsing of the XML structure of the GPX file. Upon successful reading, the application instantiates *calc*, a new instance of **Calculator**. Afterwards, it loops over all waypoints in the GPX file, using *calc* to computeVelocity(), computeCalories() and computeIncline() for each consecutive pair of waypoints. After having calculated all these metrics for every waypoint, the application instance does a self-call to generateMap(). The generated map is then used in the call to the **MapVisualiser** instance's visualiseMap() method. An alternative to this would be to visualise the map immediately after generating, however we decided against this as we wanted to avoid a scenario where a method serves more than a single purpose. After visualising the map, the instance of **MapVisualiser** is destroyed. The instance of **GPXManager** then calls createReport() on *calc*, which in turn instantiates *report* as a new instance of **Report** and returns it to the application instance. After having been created, *report* calls displayReport() to show the report to the user. Optionally, if the user chooses to generate a printable (PDF) report, the

parser does a generatePrintableReport()-call to the **Report** instance *report*, which calls saveReport() to save the report to disk. Now, the file parsing is completed.



Sequence Diagram - GUI-User Interaction



The diagram above shows how a user interacts with the GUI. First, the **GUI** instance does a self-call to drawMenuPanel(). Then, looping until the status of the file is valid, displayFilePrompt() is executed and the user gives a File path to *gui*, which calls selectFile(File path) and gets a status back by the **GPXManager** application instance. After the loop, **GPXManager** does a self-call to generateMap(). Then, looping until the activity status is "selected", *gui* does displayActivityList(), getting an Activity from the user and passing it to the application instance by calling selectActivity(Activity), which returns a status. After this loop, the **GUI** instance executes displayBiometricsPrompt(), taking Biometrics from the user. If the status returned by the application instance when the biometrics are passed to it by means of an inputBiometrics(Biometrics)-call is "Biometrics not provided", *gui* calls displayWarningMessage(). Then, the **GUI** instance calls visualiseMap() and displayReport(). Optionally, if the user chooses to save the report as a PDF, *gui* invokes generatePrintableReport(). Then, it executes displayFileSavePrompt(), getting a File path from the user and calling saveReport(File path).

Implementation

Authors: Jeroen Klaver, Khagan Mammadov, David Breitling, Lucas Ponticelli

Our initial strategy was to follow the general overview of the system diagram we came up with during our brainstorming session to start working on our implementation. This gave us a better idea of the libraries we would be using for our MVP and final product. More specifically, this helped us realise that using an external library to parse GPX files, such as JPX, was a better solution than parsing them manually. The GMapsFX library we also had previously chosen had much more desirable alternatives. We decided upon using a CLI for our MVP over a GUI to reduce the complexity of our application during the early stages of development. This is not consistent with our diagrams as we did not implement a GUI yet, however including the GUI in our diagrams helped us visualise the implementation of the GUI for our final product.

Our next strategy was to use our MVP as a starting point and expand on it when modelling our diagrams. This gave us a good idea of the basic attributes and operations needed for our application and the attributes and operations we will need for our final product. The diagrams we made provide an overview of what our incremental transition from the MVP to the final product will entail.

Ultimately, our MVP includes a CLI that handles file input, activity selection and displaying the metrics, the external JPX library to parse GPX files, an easily extensible abstract **Activity** class and a **Calculator** class that computes the final metrics using the helper classes in the **Metrics** package.

We also managed to fix a bug discussed during the meeting with our TA that would crash the application when attempting to calculate metrics for GPX files without time logs.

In our source code, the main Java class needed for executing our system is located at:
src/main/java/softwaredesign/Main.java

The Jar file for directly executing our system can be found at:
out/artifacts/software-design-vu-2020.jar

The video showing the execution of our system can be found at:
https://youtu.be/63fFLXG49tE

Time logs

Team number	2			
Member	**Activity**	**Week number**	**Hours**	
Everyone	Team meeting	3	4	1h each
Khagan, David	Implementation	3	3	1.5h each
Lucas, David	Brainstorming and General overview of the system	3	2	1h each
Lucas	Object diagram	3	3	
Khagan	Implementation	3	5	
Jeroen	Object diagram	3	2	
David	Implementation	4	1	
Lucas, Jeroen	Team meeting	4	2	1h each
David, Khagan	Team meeting	4	2	1h each
Lucas, Jeroen	Class diagram	4	4	2h each
Khagan	State machine diagrams	4	6	
David, Khagan	Sequence diagrams	4	6	3h each
David	Introduction	4	2	
Everyone	Team meeting and Textual descriptions	4	16	4h each
Everyone	Final touches	4	8	2h each
		TOTAL	66	